

YARD Architecture Reference

Copyright © 2000-2012 Brian Davis

Draft version September 2, 2012

Contents

<i>i</i>	Editorial Notes	2
1	Introduction	3
2	Implementations	3
3	Overview	3
4	Programming Model	4
	4.1 Register File	4
	4.2 Special Registers	5
	4.3 Return Stack	5
	4.4 Register Calling Conventions	5
	4.5 Input Flags	6
	4.6 Coprocessor Interface	6
5	ALU Operations	7
6	Immediates	7
7	Memory References	8
	7.1 Operand Sizing	8
	7.2 Load and Store	8
	7.3 Byte Order and Bit Numbering	8
	7.4 Data Addressing Modes	9
8	Branches & Jumps	10
	8.1 Jump Addressing	10
	8.2 Branch Addressing	10
	8.3 Delay Slots	10
9	Conditional Instructions	11
	9.1 SKIP	11
	9.2 WHEN	11
	9.3 Multi-instruction skips	11
10	Instruction Summary (Alphabetic by Mnemonic)	12
11	Instruction Encoding	15
	11.1 Logical and Move Group	15
	11.2 Arithmetic Group	16
	11.3 Memory Group	17
	11.4 Control Group	18
	11.5 Skip Condition Field	20

i Editorial Notes

Shorthand marginal notes indicating the status of a given feature can be decoded as follows:

- “??” indicates ill-defined
- “%” indicates partial implementation
- “\$\$” indicates an optional feature

Register names are presented in script: *r*₇

Instruction mnemonics within text appear in uppercase bold: **BRA**

Within assembly code examples, mnemonics appear in lowercase bold: **bra label**

1 Introduction

YARD = Yet Another RISC Design

The YARD-1 is a simple 32 bit¹ RISC architecture having 16 registers and using a compact 16 bit, two operand instruction format.

2 Implementations

A synthesizable BSD-licensed VHDL core, the Y1A, provides a rudimentary implementation of the YARD-1, and is intended for embedded FPGA designs utilizing internal block memory for code and data storage.

A portable² no-frills absolute cross assembler is provided with the Y1A core, along with an instruction set verification testbench.

3 Overview

YARD-1 Feature Summary:

- 32 bit datapath
- compact 16 bit instruction format
- 16 registers
- two operand ALU instructions: register, register | immediate
- encoded short immediates: 5 bit signed, $2^N, 2^N - 1$
- immediate prefixes:
 - IMM12 : 12 bit signed immediate
 - LDI : PC relative load of 32 bit immediate
- load/store architecture
- memory operand sizes: signed/unsigned 8/16/32 bit
- data operand addressing modes:
 - register indirect
 - register offset indirect
 - stack offset
 - synthetic PC-relative and absolute modes
- SKIP based conditionals
- PC-relative branches, absolute jumps
- one branch delay slot, with selectable null

¹Early versions had an optional 16 bit datapath configuration, which is not presently supported.

²Cross assembler written in Perl.

4 Programming Model

4.1 Register File

31		0	
	r_0		General Registers
	r_1		
	r_2		
	r_3		
	r_4		
	r_5		
	r_6		
	r_7		
	r_8		
	r_9		
	r_{10}		
	r_{11}		
	r_{12}	fp	Frame Pointer
	r_{13}	sp	Stack Pointer
	r_{14}	imm	Immediate Register
	r_{15}	pc rs	Program Counter Return Stack

Figure 1: Register File

The first twelve registers, $r_0 \dots r_{11}$, are completely general purpose.

Registers r_{12} and r_{13} , aka fp and sp , are available as base registers for the stack offset addressing mode, but are otherwise undedicated.

Register r_{14} , aka imm , is loadable via the **IMM12** and **LDI** opcodes to provide arbitrary immediate values (see section 6); this register also sources the offset field for offset indirect addressing. Otherwise, r_{14} may be used and referenced like any other general purpose register.

Register r_{15} is a dedicated register which provides access to either the current program counter or the hardware return stack:

- When used as the base register in address calculations (**LD**, **ST**, **LEA**), references to r_{15} provide the PC value of the current instruction.
- When accessed as a **MOV/LD/ST** data operand, references to r_{15} push or pop the top of the internal hardware return stack.

%%

4.2 Special Registers

??

Special registers, for processor status and control, will be eventually accessible through the system coprocessor. Likely register definitions include:

31	0	
	<i>status</i>	Status Register
	<i>control</i>	Control Register
	<i>tick_msw</i>	Clock Ticks (MSW)
	<i>tick_lsw</i>	Clock Ticks (LSW)
	<i>type</i>	Processor Type
	<i>options</i>	Processor Options (optional instructions, stack depth)
	<i>id_msw</i>	ID (MSW)
	<i>id_lsw</i>	ID (LSW)

Figure 2: Special Registers

4.3 Return Stack

A hardware return stack is used to store return addresses and processor state for subroutine calls and interrupts; stack depth is implementation specific, with a minimum depth of 16.

%%

The top of this return stack is accessible via MOV/LD/ST data operations on $r_{15} \mid rs$, allowing return addresses for non-leaf functions to be moved onto a software stack for unlimited call depth.

4.4 Register Calling Conventions

??

Preliminary, subject to change upon the existence of a compiler.

31	0	
	r_0	Function Return Value
	r_1	Parameter
	r_2	Parameter
	r_3	Parameter
	r_4	Parameter
	r_5	Callee Register
	r_6	Callee Register
	r_7	Callee Register
	r_8	Callee Register
	r_9	Caller Register (save if used)
	r_{10}	Caller Register (save if used)
	r_{11}	Caller Register (save if used)
	$r_{12} \mid fp$	Caller Register (save if used) Frame Pointer
	$r_{13} \mid sp$	Stack Pointer
	$r_{14} \mid imm$	Immediate Register, used by compiler and assembler to build arbitrary constants

Figure 3: Calling Convention

4.5 Input Flags

The processor includes 16 hardware input flags, which can be individually tested using the skip-on-flag instructions:

```
skip.fs  #N      ; SKIP Flag Set
skip.fc  #N      ; SKIP Flag Clear
```

4.6 Coprocessor Interface

??

The coprocessor interface has not been fully defined; one opcode has been reserved for coprocessor operations, along with a skip slot for coprocessor conditionals.

Provision exists for up to 16 coprocessors, with the first eight intended for system and standardized usage (control, memory, math, etc.), and the last eight available for user extensions.

The coprocessors have read visibility into the current register file state, allowing coprocessor calls to resemble normal function calls.

<i>system</i>	0 : System Control
<i>mmu</i>	1 : Memory Management Unit
—	2 : reserved
—	3 : reserved
<i>imath</i>	4 : Integer Math
—	5 : reserved
<i>fpu</i>	6 : Floating Point Unit
—	7 : reserved
<i>user</i>	8 : User defined
<i>user</i>	9 : User defined
<i>user</i>	10 : User defined
<i>user</i>	11 : User defined
<i>user</i>	12 : User defined
<i>user</i>	13 : User defined
<i>user</i>	14 : User defined
<i>user</i>	15 : User defined

Figure 4: Coprocessors

5 ALU Operations

ALU operations have the general form:

$$r_a \leftarrow r_a \text{ } OP \text{ } op_b \quad \text{written as } OP \text{ } r_a, \text{ } op_b \text{ in assembly code}$$

Where r_a is a register, and op_b is either a register r_b , or a short immediate constant (the short encodings accomodate 5 bit signed, 2^N , and $2^N - 1$ constants).

Selectable inversion of the B operand is provided for the logical instructions, allowing for convenient bit mask generation when used with the 2^N and $2^N - 1$ encoded short immediates.

<i>mnemonic</i>	<i>name</i>
<code>mov{.not}</code>	MOVE
<code>and{.not}</code>	AND
<code>or{.not}</code>	OR
<code>xor{.not}</code>	eXclusive OR
<code>add</code>	ADD
<code>sub</code>	SUBtract
<code>rsub</code>	Reverse SUBtract
<code>lsr</code>	Logical Shift Right
<code>lsl</code>	Logical Shift Left
<code>asr</code>	Arithmetical Shift Right
<code>ror</code>	ROtate Right
<code>rol</code>	ROtate Left
<code>flip</code>	bit FLIP
<code>ff1,cnt1</code>	(moving to coprocessor space)

Table 1: ALU Instructions

Note that in assembly code, the destination register operand r_a comes first:

```

mov    r0,r1    ; r0 = r1
add    r0,r4    ; r0 = r0 + r4
sub    r0,r4    ; r0 = r0 - r4
skip.gt r0,r4   ; r0 > r4
```

6 Immediates

Provision for immediates beyond the reach of the op_b short encodings is available via r_{14} , the *imm* register. *Imm* may be loaded either via any normal register operation, or by using one of the two dedicated immediate instructions:

- `imm12 #CONSTANT ; IMMEDIATE, 12 bit`
Loads a 12 bit sign-extended value into *imm*
- `ldi LABEL ; Load IMMEDIATE`
Loads a PC Relative quad (0..4095 quad offset) into *imm*

??

A future version of the assembler will support an IMM macro to load *imm* using the most compact encoding method (encoded short immediate MOV, IMM12, or LDI).

```
imm    #CONSTANT    ; IMMEDIATE, select best encoding
```


7 Memory References

7.1 Operand Sizing

Memory references support three operand sizes :

- byte = 8 bits
- wyde³ = 16 bits
- quad = 32 bits

7.2 Load and Store

Memory is byte addressable, with address alignment **REQUIRED** for 16/32 bit operands.

LD allows sign or zero extended 8/16/32 bit memory reads.

ST allows 8/16/32 bit memory writes.

7.3 Byte Order and Bit Numbering

Byte order in memory is Big-Endian, having the most significant byte of a multi-byte operand stored at the lowest memory address.

Bit numbering in registers, buses, and other multi-bit fields is Little-Bittian, wherein an N bit field has the MSB numbered N-1, and the LSB numbered zero; thus bit number N has a numerical weight of 2^N .

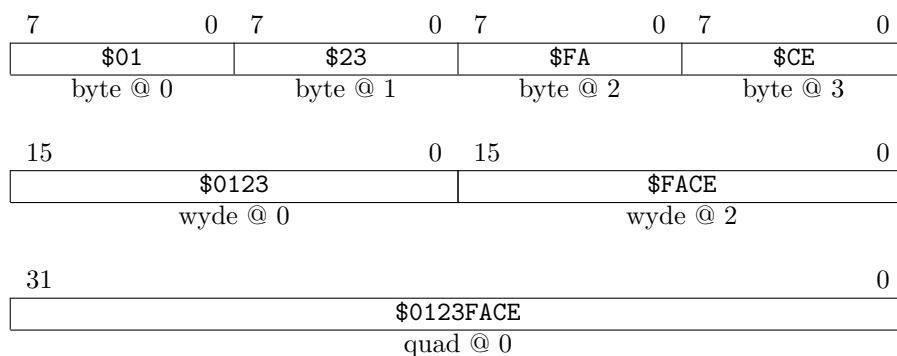


Figure 5: Operands in Memory

³per Knuth"MMIXware",1999,p.4: "Weight Watchers know that two nybbles make one byte, but two bytes make one wyde"

7.4 Data Addressing Modes

Fundamental Addressing Modes

- Register Indirect: (Ra)

```
ld    r0, (r4)
```

The effective address is sourced by register r_a .

- Register Offset Indirect: .imm(Ra)

```
imm12  #$3E4  
ld    r0, .imm(r4)
```

The effective address is formed from the sum $r_a + imm$.

A prefix instruction (IMM12 or LDI), or other register operation, is needed to first load imm .

??

Once the assembler supports the automatic generation of constant prefixes, the offset will be able to be supplied directly in the memory reference:

```
ld    r0, some_offset(r4)
```

- Stack Offset: stk_offset(fp | sp)

```
ld.q   r0, 48(sp)
```

This addressing mode supports **ONLY** quad loads and stores.

The sp and fp registers **MUST** be quad aligned when using this address mode.

The stack offset is encoded directly in the instruction as a 4 bit field, yielding an offset range of 0-15 quads (in assembly code, `stk_offset` is a byte offset with an allowed range of 0,4 ... 60 bytes); offsets outside this range should be assembled as a normal register offset indirect access.

Synthetic Addressing Modes

The use of register offset indirect addressing with pc and imm as the base register allows PC relative and absolute address modes to be synthesized, in both short (IMM12 prefix) and long (LDI prefix) variants.

As the IMM12 prefix is signed, 'short absolute' mode can reach the top and bottom 2K bytes of memory, which works well for memory and I/O in tiny embedded systems.

??

The assembler does **NOT** currently support automatic generation of these addressing modes, but when it does the syntax will look something like this:

- PC Relative: label(pcr)

```
lea    r0, some_label(pcr)
```

- Absolute: address

```
st.b   r0, $ffff_ff00
```

8 Branches & Jumps

8.1 Jump Addressing

The jump instructions (JMP/JSR) use a 32 bit absolute address in a register, e.g. `jmp (r1)`.

8.2 Branch Addressing

The branch instructions support PC relative addressing in both short and long variants, with instruction (16 bit) offsets of 9, 21, and 31 bits:

<i>instruction(s)</i>	<i>offset size</i>	<i>branch range, instructions</i>
<code>bra bsr</code>	9 bit	$+255 \dots - 256$
<code>imm12 + lbra lbsr</code>	21 bit	$+1,048,575 \dots - 1,048,576$
<code>ldi + lbra lbsr</code>	31 bit	$+2^{30} - 1 \dots - 2^{30}$
<code>rbra rbsr</code>	31 bit	$+2^{30} - 1 \dots - 2^{30}$

Table 2: Branch Offsets

The long branch variants use an IMM12 or LDI prefix to load *imm* with the upper bits of the branch offset, which is combined with 9 bits from the branch instruction field to form the 21 or 31 bit instruction offset.

8.3 Delay Slots

The programming model exposes one delay slot for the change-of-flow instructions; execution of the delay slot instruction is enabled by the `.d` suffixed instruction variants, e.g. `BRA.D`, `JSR.D`

9 Conditional Instructions

9.1 SKIP

Conditional execution is supported via the **SKIP** instruction, which nullifies the execution of the following instruction if the skip condition is true.

Supported **SKIP** conditions (see SCC table in section 11.5) include register-register, register-zero, and assorted bit/byte/wyde tests; the **ADD/SUB/RSUB** instructions also provide a skip-no-carry/borrow mode.

Conditional change-of-flow is obtained by skipping a branch or jump.

9.2 WHEN

An alternate mnemonic for skip, **WHEN**, generates a skip of the opposite condition sense ⁴ This provides conditional execution of the following instruction when the **WHEN** condition is true.

9.3 Multi-instruction skips

%%

The **SPAM** [**Skip Propagate Against Mask**] instruction, when placed in the shadow of a skip, propagates the skip test result to the following eight instructions using either **AND** or **XOR-NOT** masking modes, whereby any given instruction is skipped if the corresponding bit of the resulting skip vector is set.

SPAM in **AND** mode [$skip_n \leftarrow skip_test \text{ AND } skip_mask_n$] skips the masked instructions when the skip condition is true, and always executes the non-masked instructions, thus extending the basic skip operation to multiple instructions (which may be intertwined with non-skipped instructions).

SPAM in **XOR-NOT** mode [$skip_n \leftarrow (skip_test \text{ XOR-NOT } skip_mask_n) \text{ AND } trunc_mask_n$] implements a branch free construct akin to an if-else for short instruction sequences, in which only one of the two sets of instructions is executed. *Trunc_mask* allows the **XOR-NOT** mode sequence length to be shortened to fewer than eight instructions.

⁴**WHEN** was inspired by the NIOS I IFS, which performed the same conditional inversion for the NIOS **SKPS** instruction

10 Instruction Summary (Alphabetic by Mnemonic)

Opcode	Mnemonic	Name

01000ttbbbbaaaa	add	ADD
01001ttbbbbaaaa	add.snc	ADD, Skip No Carry
00010ttbbbbaaaa	and	AND
00011ttbbbbaaaa	and.not	AND NOT
0111010bbbbaaaa	asr	Arithmetical Shift Right
1110001rrrrrrrrr	bra	BRANch
1110000rrrrrrrrr	bra.d	BRANch, Delayed
1110011rrrrrrrrr	bsr	Branch SubRoutine
1110010rrrrrrrrr	bsr.d	Branch SubRoutine, Delayed
000000100000aaaa	clr	CLear
010100100001aaaa	dec	DECrement
01111111bbbbaaaa	ext.sb	EXTend, Signed Byte
01111101bbbbaaaa	ext.sw	EXTend, Signed Wyde
01111110bbbbaaaa	ext.ub	EXTend, Unsigned Byte
01111100bbbbaaaa	ext.uw	EXTend, Unsigned Wyde
0111011bbbbaaaa	flip	FLIP
1011111111111111	imm12	IMMediate, 12 bit
010000100001aaaa	inc	INCrement
111110100000aaaa	jmp	JuMP
111110000000aaaa	jmp.d	JuMP, Delayed
111111100000aaaa	jsr	Jump SubRoutine
111111000000aaaa	jsr.d	Jump SubRoutine, Delayed
1110101rrrrrrrrr	lbra	Long BRANch
1110100rrrrrrrrr	lbra.d	Long BRANch, Delayed
1110111rrrrrrrrr	lbsr	Long Branch SubRoutine
1110110rrrrrrrrr	lbsr.d	Long Branch SubRoutine, Delayed
1000mss0bbbbaaaa	ld	LoaD
1000mss1bbbbaaaa	ld.b	LoaD, Byte
1000mss0bbbbaaaa	ld.q	LoaD, Quad
1000mss1bbbbaaaa	ld.sb	LoaD, Signed Byte
1000mss1bbbbaaaa	ld.sw	LoaD, Signed Wyde
1000mss0bbbbaaaa	ld.ub	LoaD, Unsigned Byte
1000mss0bbbbaaaa	ld.uw	LoaD, Unsigned Wyde
1000mss1bbbbaaaa	ld.w	LoaD, Wyde
1010rrrrrrrrrrrrr	ldi	LoaD Immediate
1001mss1bbbbaaaa	lea	Load Effective Address
0111001bbbbaaaa	lsl	Logical Shift Left
0111000bbbbaaaa	lsr	Logical Shift Right
00000ttbbbbaaaa	mov	MOVE
00001ttbbbbaaaa	mov.not	MOVE NOT
011000100000aaaa	neg	NEGate
0000000000000000	nop	No OPeration
001100111111aaaa	not	NOT
00100ttbbbbaaaa	or	OR
00101ttbbbbaaaa	or.not	OR NOT
0111101bbbbaaaa	rol	Rotate Left
0111100bbbbaaaa	ror	Rotate Right
01100ttbbbbaaaa	rsub	Reverse SUBtract
01101ttbbbbaaaa	rsub.snb	Reverse SUBtract, Skip No Borrow

Opcode	Mnemonic	Name
1111111000100000	rti	ReTurn from Interrupt
1111110000100000	rti.d	ReTurn from Interrupt, Delayed
1111101000100000	rts	ReTurn from Subroutine
1111100000100000	rts.d	ReTurn from Subroutine, Delayed
1101010000000000	skip	SKIP
1101010000000000	skip.a	SKIP Always
110101010110aaaa	skip.abm	SKIP Any Byte Minus
110101010010aaaa	skip.abz	SKIP Any Byte Zero
110101010101aaaa	skip.awm	SKIP Any Wyde Minus
110101010001aaaa	skip.awz	SKIP Any Wyde Zero
1101111bbbbbaaaa	skip.bc	SKIP Bit Clear
1101011bbbbbaaaa	skip.bs	SKIP Bit Set
11010100bbbbbaaaa	skip.eq	SKIP Equal
110111010111aaaa	skip.fc	SKIP Flag Clear
110101010111aaaa	skip.fs	SKIP Flag Set
11011010bbbbbaaaa	skip.ge	SKIP Greater than or Equal
11011011bbbbbaaaa	skip.gt	SKIP Greater Than
110111010100aaaa	skip.gtz	SKIP Greater than Zero
11011001bbbbbaaaa	skip.hi	SKIP Higher
11011000bbbbbaaaa	skip.hs	SKIP Higher or Same
11010011bbbbbaaaa	skip.le	SKIP Less than or Equal
110101010100aaaa	skip.lez	SKIP Less than or Equal Zero
11010000bbbbbaaaa	skip.lo	SKIP Lower
11010001bbbbbaaaa	skip.ls	SKIP Lower or Same
11010010bbbbbaaaa	skip.lt	SKIP Less Than
110101111111aaaa	skip.mi	SKIP Minus
1101110000000000	skip.n	SKIP Never
110111010110aaaa	skip.nbm	SKIP No Byte Minus
110111010010aaaa	skip.nbz	SKIP No Byte Zero
11011100bbbbbaaaa	skip.ne	SKIP Not Equal
110111010101aaaa	skip.nwm	SKIP No Wyde Minus
110111010001aaaa	skip.nwz	SKIP No Wyde Zero
110111010000aaaa	skip.nz	SKIP NonZero
110111111111aaaa	skip.pl	SKIP Plus
110101010000aaaa	skip.z	SKIP Zero
11110nnnnmmmmmmmm	spam.and	Skip Propagate Against Mask, AND mode
11110nnnnmmmmmmmm	spam.xorn	Skip Propagate Against Mask, XOR-Not mode
1001mss0bbbbbaaaa	st	STore
1001mss0bbbbbaaaa	st.b	STore, Byte
1001mss0bbbbbaaaa	st.q	STore, Quad
1001mss0bbbbbaaaa	st.w	STore, Wyde
01010ttbbbbbaaaa	sub	SUBtract
01011ttbbbbbaaaa	sub.snb	SUBtract, Skip No Borrow
1101110000000000	when	WHEN
1101110000000000	when.a	WHEN Always
110111010110aaaa	when.abm	WHEN Any Byte Minus
110111010010aaaa	when.abz	WHEN Any Byte Zero
110111010101aaaa	when.awm	WHEN Any Wyde Minus
110111010001aaaa	when.awz	WHEN Any Wyde Zero
1101011bbbbbaaaa	when.bc	WHEN Bit Clear
1101111bbbbbaaaa	when.bs	WHEN Bit Set
11011100bbbbbaaaa	when.eq	WHEN Equal

Opcode	Mnemonic	Name

110101010111aaaa	when.fc	WHEN Flag Clear
110111010111aaaa	when.fs	WHEN Flag Set
11010010bbbbaaaa	when.ge	WHEN Greater than or Equal
11010011bbbbaaaa	when.gt	WHEN Greater Than
110101010100aaaa	when.gtz	WHEN Greater than Zero
11010001bbbbaaaa	when.hi	WHEN Higher
11010000bbbbaaaa	when.hs	WHEN Higher or Same
11011011bbbbaaaa	when.le	WHEN Less than or Equal
110111010100aaaa	when.lez	WHEN Less than or Equal Zero
11011000bbbbaaaa	when.lo	WHEN Lower
11011001bbbbaaaa	when.ls	WHEN Lower or Same
11011010bbbbaaaa	when.lt	WHEN Less Than
110111111111aaaa	when.mi	WHEN Minus
1101010000000000	when.n	WHEN Never
110101010110aaaa	when.nbm	WHEN No Byte Minus
110101010010aaaa	when.nbz	WHEN No Byte Zero
11010100bbbbaaaa	when.ne	WHEN Not Equal
110101010101aaaa	when.nwm	WHEN No Wyde Minus
110101010001aaaa	when.nwz	WHEN No Wyde Zero
110101010000aaaa	when.nz	WHEN NonZero
110101111111aaaa	when.pl	WHEN Plus
110111010000aaaa	when.z	WHEN Zero
00110ttbbbbbaaaa	xor	eXclusive OR
00111ttbbbbbaaaa	xor.not	eXclusive OR NOT

DIRECTIVE	.verify	.VERIFY
DIRECTIVE	align	ALIGN
DIRECTIVE	end	END
DIRECTIVE	equ	EQUate
DIRECTIVE	org	ORiGin
DIRECTIVE	dc.b	Define Constant, Byte
DIRECTIVE	dc.q	Define Constant, Quad
DIRECTIVE	dc.s	Define Constant, String
DIRECTIVE	dc.w	Define Constant, Wyde
DIRECTIVE	dc.z	Define Constant, Zero terminated string

11 Instruction Encoding

11.1 Logical and Move Group

15:12	11	10:9	8:4	3:0	Operation			
0000	NOTB	OPB_CTL	OPB	RA	MOV	RA	=	{NOT} OPB
0001	NOTB	OPB_CTL	OPB	RA	AND	RA	= RA	AND {NOT} OPB
0010	NOTB	OPB_CTL	OPB	RA	OR	RA	= RA	OR {NOT} OPB
0011	NOTB	OPB_CTL	OPB	RA	XOR	RA	= RA	XOR {NOT} OPB

Abbreviations:

RA : 4 bit register field (source 1 & destination)

OPB : 5 bit immediate/4 bit register field (source 2)

OPB_CTL : select OPB type
00: register
01: 5 bit sign extended immediate
10: 2^N immediate
11: $2^N - 1$ immediate

NOTB : if set, use NOT OPB

11.2 Arithmetic Group

15:12	11	10:9	8:4	3:0	Operation			

0100	CSKIP	OPB_CTL	OPB	RA	ADD	RA	=	RA + OPB
0101	CSKIP	OPB_CTL	OPB	RA	SUB	RA	=	RA - OPB
0110	CSKIP	OPB_CTL	OPB	RA	RSUB	RA	=	OPB - RA
15:12	11:10	9	8:4	3:0	Operation			

0111	00	0	N	RA	LSR	RA	=	RA LSR N
0111	00	1	N	RA	LSL	RA	=	RA LSL N
0111	01	0	N	RA	ASR	RA	=	RA ASR N
0111	01	1	N	RA	FLIP(*)	RA	=	RA FLIP N
0111	10	0	N	RA	ROR	RA	=	RA ROR N
0111	10	1	N	RA	ROL	RA	=	RA ROL N
15:12	11:10	9:8	7:4	3:0	Operation			

0111	11	00	RB	RA	EXT.UW	RA	=	zero extend RB[15:0]
0111	11	01	RB	RA	EXT.SW	RA	=	sign extend RB[15:0]
0111	11	10	RB	RA	EXT.UB	RA	=	zero extend RB[7:0]
0111	11	11	RB	RA	EXT.SB	RA	=	sign extend RB[7:0]

Note : Bit Counting instructions (CNT1,FF1) are moving to coprocessor space

Abbreviations:

RA : 4 bit register field (source 1 & destination)
 RB : 4 bit register field (source 2)
 OPB : 5 bit immediate/4 bit register field (source 2)

OPB_CTL : select OPB type
 00: register
 01: 5 bit sign extended immediate
 10: 2^N immediate
 11: 2^N - 1 immediate

CSKIP : if set, skip the next instruction if NO carry/borrow occurred
 if cleared, normal execution of following instruction

N : 5 bit field
 - bit count for shift and rotate instructions

 - bit swap enable for FLIP
 N(4) : swap even/odd wydes
 N(3) : swap even/odd bytes
 N(2) : swap even/odd nybbles
 N(1) : swap even/odd bit pairs
 N(0) : swap even/odd bits
 e.g.
 11000 : byte reverse register (swap wydes & bytes)
 11111 : bit reverse register (swap everything)
 00111 : bit reverse all bytes in register

⁴universal bit reverse per H. Warren, "Hacker's Delight", 2003, page 102, FLIP instruction credited to Guy Steele

11.3 Memory Group

15:12	11	10:9	8	7:4	3:0	Operation	
1000	MODE	SIZE	SIGN	RB	RA	LD	RA = {extend} [RB {+ IMM }]
1001	MODE	SIZE	0	RB	RA	ST	[RB {+ IMM }] = {trunc} RA
1001	MODE	SIZE	1	RB	RA	LEA	RA = RB { + IMM }

15:12					11:0	Operation	
1010					EA12	LDI	Load Immediate IMM = [EA12*4 + PC & \$FFFF_FFFC]

15:12					11:0	Operation	
1011					I12	IMM12	IMM = sign extend I12

Abbreviations:

RA : 4 bit register field (destination)

RB : 4 bit register field (address)

SIGN : when set, sign extend memory operand on load

SIZE : load/store operand size

00 quad (32 bit) stack offset addressing

01 quad (32 bit)

10 wyde (16 bit)

11 byte (8 bit)

MODE : 0 = no offset, 1 = use IMM register for normal addressing modes
0 = fp 1 = sp for stack offset addressing mode

I12 : signed 12 bit immediate for IMM12

EA12 : unsigned 12 bit quad offset for LDI

11.4 Control Group

15:12	11	10:9	8	7:4	3:0	Operation
1100	?	?	?	?	?	CP coprocessor

15:12	11:0	Operation
1101	SCC	SKIP.CC if (cond) skip next instruction

15:12	11	10	9	8:0	Operation
1110	LBRA	PSHPC	DNUL	I9	{L}BRA/{L}BSR { push PC };

15:12	11	10:8	7:0	Operation	PC = PC + ({IMM << 10} + I.9) * 2
1111	0	TRUNC	MASK	SPAM.AND XOR	skip propagate

15:12	11	10	9	8	7:4	3:0	Operation
1111	1	PSHPC	DNUL	0	0000	RA	JMP/JSR { push PC }; PC = RA
1111	1	PSHPC	DNUL	0	0001	RA	RBRA/RBSR { push PC }; PC = PC + RA
1111	1	POPSR	DNUL	0	0010	0000	RTS/RTI pop PC; { pop SR }

15:12	11	10	9	8	7:0	Operation
1111	1	1	DNUL	1	TR	TRAP push PC,SR; PC = vector(TR)

Abbreviations:

RA : 4 bit register field (source 1)

DNUL : 1 = nullify delay slot instruction
0 = execute delay slot instruction

PSHPC : push PC

POPSR : pop SR (used to turn an RTS into an RTI)

LBRA : enables long branch: augments IMM9 offset with IMM << 10

I9 : signed nine bit instruction offset for BRA
(for LBRA, I9 is the low 9 bits of a 21 | 31 bit signed offset)

SCC : skip condition field, see table in next section

TRAP : 8 bit trap vector

MASK : skip propagate mask
1 = skip enable
D7 = next instruction .. D0 = eighth instruction

TRUNC : skip propagate truncate field
7 : AND mode
6..0 : XOR NOT mode, truncate skip mask by N instructions

11.5 Skip Condition Field

SCC : skip condition

11 10:8 7:4 3:0

0	000	RB	RA	skip.lo	lower	unsigned, RA < RB
1	000	RB	RA	skip.hs	higher or same	unsigned, RA >= RB
0	001	RB	RA	skip.ls	lower or same	unsigned, RA <= RB
1	001	RB	RA	skip.hi	higher	unsigned, RA > RB
0	010	RB	RA	skip.lt	less than	signed, RA < RB
1	010	RB	RA	skip.ge	greater than or equal	signed, RA >= RB
0	011	RB	RA	skip.le	less than or equal	signed, RA <= RB
1	011	RB	RA	skip.gt	greater than	signed, RA > RB
0	100	RB	RA	skip.eq	equal	RA == RB
1	100	RB	RA	skip.ne	not equal	RA != RB
0	101	0000	RA	skip.z	zero	RA == 0
1	101	0000	RA	skip.nz	non-zero	RA != 0
0	101	0001	RA	skip.awz	any wyde zero	
1	101	0001	RA	skip.nwz	no wyde zero	
0	101	0010	RA	skip.abz	any byte zero	
1	101	0010	RA	skip.nbz	no byte zero	
0	101	0011	RA		reserved	
1	101	0011	RA		reserved	
0	101	0100	RA	skip.lez	less than or equal zero	RA <= 0
1	101	0100	RA	skip.gtz	greater than zero	RA > 0
0	101	0101	RA	skip.awm	any wyde minus	
1	101	0101	RA	skip.nwm	no wyde minus	
0	101	0110	RA	skip.abm	any byte minus	
1	101	0110	RA	skip.nbm	no byte minus	
0	101	0111	N	skip.fs	flag N set	test input flag bits
1	101	0111	N	skip.fc	flag N clear	
0	101	1ccc	N	skip.cpt	coprocessor N true	
1	101	1ccc	N	skip.cpf	coprocessor N false	

(continued on next page)

Aliased skips:

skip plus/minus aliases to "skip.bc/bs rn, #31" :

11:0

11111111aaaa	skip.pl	plus	RA >= 0
01111111aaaa	skip.mi	minus	RA < 0

skip always/never aliases to "skip.eq/ne r0,r0" :

11:0

010000000000	skip.a	always
110000000000	skip.n	never

Skip on bit:

11 10:9 8:4 3:0

0	11	B	RA	skip.bs	bit RA.B set
1	11	B	RA	skip.bc	bit RA.B clear

Abbreviations:

RA : 4 bit register field (source 1)
RB : 4 bit register field (source 2)

N : 4 bit input flag or coprocessor number

B : bit number (for "skip on bit")

ccc : coprocessor skip type